1. Introduction to the C++ Toolkit

Created: July 1, 2003
Updated: January 29, 2004

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

One difficulty in understanding a major piece of software such as the C++ Toolkit is knowing where to begin in understanding the overall framework or getting the 'big picture' of how all the different components relate to each other. One approach is to dive into the details of one component and understand it in sufficient detail to get a roadmap of the rest of the components, and then repeat this process with the other components. Without a formal road map, this approach can be very time consuming and it may take a long time to locate the functionality one needs.
When trying to understand a major piece of software, it would be more effective if there is a written text that explains the overall framework without getting too lost in the details. This chapter is written with the intent to provide you with this broader picture of the C++ Toolkit.
This chapter provides an introduction to the major components that make up the toolkit. You can use this chapter as a roadmap for the rest of the chapters that follow.

Chapter Outline

The following is an outline of the topics presented in this chapter:

- The CORELIB Module
    - Application Framework
    - Argument processing
    - Diagnostics
    - Environment Interface
    - Files and Directories
    - MT Test wrappers

    - Object and Ref classes
    - Portability definitions
    - Portable Exception Handling
    - Portable Process Pipes
    - Registry
    - STL Use Hints
    - Stream Wrappers
    - String Manipulations
    - Template Utilities
    - Threads
    - Time
- The ALGORITHM Module
- The CGI Module
- The CONNECT Module
    - Socket classes
    - Connector and Connection Handles
    - Connection Streams
    - Sendmail API
    - Threaded Server
- The CTOOL Module
- The DBAPI Module
    - Database User Classes
    - Database Driver Architecture
- The GUI Module

- The HTML Module
    - Relationships between HTML classes
    - HTML Processing
- The OBJECT MANAGER Module
- The SERIAL Module
    - The Input Stream Class
    - The Output Stream Class
    - The Hook Classes
- The UTIL Module
    - Checksum
    - Console Debug Dump Viewer
    - Lightweight Strings
    - Range Support
    - Weak Map Templates
    - Linked Sets
    - Random Number Generator
    - Registry based DNS
    - Resizing Iterator
    - Rotating Log Streams
    - Stream Support
    - String Search
    - Thread Pools
    - UTF 8 Conversion

The CORELIB Module

The C++ Toolkit can be seen as consisting of several major pieces of code that we will refer to as *module*. The core module is called, appropriately enough, CORELIB, and provides a portable way to write C++ code and many useful facilities such as an application framework, argument processing, template utilities, threads, etc. The CORELIB facilities are used by other major modules. The rest of the sections that follow discusses the CORELIB and the other C++ Toolkit modules in more detail.

The following is a list of the CORELIB facilities. Note that each facility may be implemented by a number of C++ classes spread across many files.

- Application Framework
- Argument processing
- Diagnostics
- Environment Interface
- Files and Directories
- MT Test wrappers
- Object and Ref classes
- Portability definitions
- Portable Exception Handling
- Portable Process Pipes
- Registry
- STL Use Hints
- Stream Wrappers
- String Manipulations
- Template Utilities
- Threads
- Time

A brief description of each of each of these facilities are presented in the subsections that follow:

Application Framework

The Application framework primarily consists of an abstract class called ***CNcbiApplication*** which defines the high level behavior of an application. For example, every application upon loading seems to go through a cycle of doing some initialization, then some processing, and upon completion of processing, doing some clean up activity before exiting. These three phases are modeled in the ***CNcbiApplication*** class as interface methods ***Init()***, ***Run()***, and ***Exit()***.

A new application is written by deriving a class from the ***CNcbiApplication*** base class and writing an implementation of the ***Init()***, ***Run()***, and ***Exit()*** methods. Execution control to the new application is passed by calling the application object's ***AppMain()*** method inherited from the ***CNcbiApplication*** base class (see Figure 1). The ***AppMain()*** method is similar to the ***main()*** method used in C/C++ programs and calls the ***Init()***, ***Run()***, and ***Exit()*** methods.

More details on the use of ***CNcbiApplication*** class are presented in a later chapter.

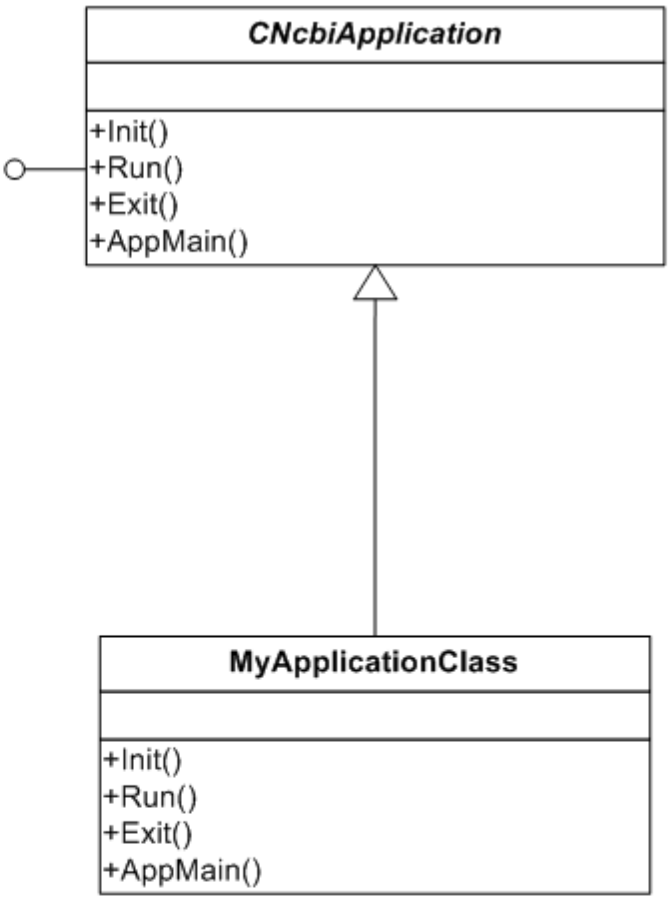

Figure 1: The ***CNcbiApplication*** class

Argument processing

In a C++ program, control is transferred from the command line to the program via the ***main()*** function. The ***main()*** function is passed a count of the number of arguments (int argc), and an array of character strings containing arguments to the program (`char** argv`). As long as the argument types are simple, one can simply set up a loop to iterate through the array of argument values and process them. However, with time applications evolve and grow more complex. Often there is a need to do some more complex argument checking. For example, the application may want to enforce a check on the number and position of arguments, check the argument type (int, string, etc.), check for constraints on argument values, check for flags, check for arguments that follow a keyword (***-logfile mylogfile.log***), check for mandatory arguments, display usage help on the arguments, etc.

To make the above tasks easier, the CORELIB provides a number of portable classes that encapsulate the functionality of argument checking and processing. The main classes that provide this functionality are the ***CArgDescriptions***, ***CArgs***, ***CArgValue*** classes.

Argument descriptions such as the expected number, type, position, mandatory and optional attributes are setup during an application's initilization such as the application object's ***Init()*** method (see previous section) by calling the ***CArgDescriptions*** class methods. Then, the arguments are extracted by calling the ***CArgs*** class methods.

More details on the argument processing are presented in a later chapter.

Diagnostics

It is very useful for an application to post messages about its internal state or other diagnostic information to a file, console or for that matter any output stream. The CORELIB provides a portable diagnostics facility that enables an application to post diagnostic messages of various severity levels to an output stream. This diagnostic facility is provided by the CNcbiDiag class. You can set the diagnostic stream to the standard error output stream (`NcbiErr`) or to any other output stream.

You can set the serverity level of the message to Information, Warning, Error, Critical, Fatal, or Trace. You can alter the severity level at any time during the use of the diagnostic stream.

More details on diagnostic streams and processing of diagnostic messages is presented in a later chapters.

Environment Interface

An application can read the environment variable settings (such as PATH) that are in affect when the application is run. CORELIB defines a portable ***CNcbiEnvironment*** class that stores the environment variable settings and provides applications with methods to get the environment variable values.

More details on the environment interface are presented in a later chapter.

Files and Directories

An application may need access to information about a file or directory. The CORELIB provides a number of portable classes to model a system file and directory. Some of the important classes are ***CFile*** for modeling a file, ***CDir*** for modeling a directory, and ***CMemoryFile*** for memory mapped file.

For example, if you create a ***CFile*** object corresponding to a system file, you can get the file's attribute settings such as file size, permission settings, or check the existence of a file. You can get the directory where the file is located, the base name of the file, and the file's extension. There are also a number of useful functions that are made available through these classes to parse a file path or build a file path from the component parts such as a directory, base name, and extension.

More details on file and directory classes is presented in a later chapters.

MT Test wrappers

The CNcbiApplication class which was discussed earlier provides a framework for writing portable applications. For writing portable multi-threaded applicatioms, the CORELIB provides a ***CThreadedApp*** class derived from ***CNcbiApplication*** class which provides a framework for building multi-threaded applications.

Instead of using the Init(), Run, Exit() methods for the ***CNcbiApplication*** class, the ***CThreadedApp*** class defines specialized methods such as ***Thread_Init()***, ***Thread_Run()***, ***Thread_Exit()***, ***Thread_Destroy()*** for controlling thread behavior. These methods operate on a specific thread identified by a thread index parameter.

Object and Ref classes

A major cause of errors in C/C++ programs is due to dynamic allocation of memory. Stated simply, memory for objects allocated using the new operator must be released by a corresponding delete operator. Failure to delete allocated memory results in memory leaks. There may also be programming errors caused by references to objects that have never been allocated or improperly allocated. One reason these types of problems crop up are because a programmer may dynamically allocate memory as needed, but may not deallocate it due to unanticipated execution paths.

The C++ standard provides the use of a template class, auto_ptr , that wraps memory management inside constructors and destructors. Because a destructor is called for every constructed object, memory allocation and deallocation can be kept symmetrical with respect to each other. However, the auto_ptr does not properly handle the issue of ownership when multiple auto pointers, point to the same object. What is needed is a reference counted smart pointer that keeps a count of the number of pointers pointing to the same object. An object can only be released when its reference count drops to zero.

The CORELIB implements a portable reference counted smart pointer through the ***CRef*** and ***CObject*** classes. The ***CRef*** class provides the interface methods to access the pointer and the ***CObject*** is used to stores the object and the reference count.

More ***CObject*** classes are presented in a later chapter.

Portability definitions

To help with portability, the CORELIB uses only those C/C++ standard types that have some guarantees about size and representation. In particular, use of long, long long, float is not recommended for portable code.

To help with portability, integer types such as `Int1`, `Uint1`, `Int2`, `Uint2`, `Int4`, `Uint4`, `Int8`, `Uint8` have been defined with constant limits. For example, a signed integer of two bytes size is defined as type `Int2` with a minimum size of `kMin_I2` and a maximum size of `kMax_I2`. There are minimum and maximum limit constants defined for each of the different integer types.

More details on standard portable data types are presented in a later chapter.

Portable Exception Handling

C++ defines a structured exception handling mechanism to catch and process errors in a block of code. When the error occurs an exception is thrown and caught by an exception handler. The exception handler can then try to recover from the error, or process the error. In the C++ standard, there is only one exception class (std::exception), that stores a text message that can be printed out. The information reported by the std::exception may not be enough for a complex system. The CORELIB defines a portable ***CException*** class derived from std::exception class that remedies the short comings of the standard exception class

The CORELIB defines a portable ***CException*** class derived from std::exception class. The ***CException*** class in turn serves as a base class for many other exception classes specific to an application area such as the ***CCoreException***, ***CAppException***, ***CArgException***, ***CFileException***, and so on. Each of these derived classes can add facilities specific to the application area they deal with.

These exception classes provides many useful facilties such as a unique identification for every exception that is thrown, the location (file name and line number) where the exception occurred, references to lower-level exceptions that have already been thrown so that a more complete picture of the chain of exceptions is available, ability to report the exception data into an arbitrary output channel such as a diagnostic stream, and format the message differently for each channel.

More details on exceptions and exception handling are presented in a later chapter.

Portable Process Pipes

A pipe is a common mechanism used to establish communications between two separate processesis. The pipe serves as a communication channel between processes.

The CORELIB defines the ***CPipe*** class that provides a portable inter-process cimmunications facility between a parent process and its child process. The pipe is created by specifying the command and arguments used to start the child process and specifying the type of data channels (text or binary) that will connect the processes. Data is sent acrross the pipe using the ***CPipe*** read and write methods.

Registry

The settings for an application are often read from a configuration or initialization file. This configuration file may define the parameters needed by the application. For example, many Unix programs read their parameter settings from configuration files. Similarly, Windows programs may read and store information in an internal registry database, or an initialization file.

The NcbiRegistry class provides a portable facility to access, modify and store runtime information read from a configuration file. The configuration file consists of sections. A section is defined by a section header of the form [***section-header-name***]. Within each section, the parameters are defined using (name, value) pairs and represented as ***name=value*** strings. The syntax closely resembles the '.ini' files used in Windows and also by Unix tools such as Samba.

More details on the Registry are presented in a later chapter.

STL Use Hints

To minimize naming conflicts, all NCBI code is placed in the ncbi name space. The CORELIB defines a number of portable macros to help manage name space definitions. For example, you can use the `BEGIN_NAME_SPACE` macro at the start of a section of code to place that code in the specified name space. The `END_NAME_SPACE` macros is used to indicate the end the of the name space definition. To declare the use of the NCBI namespace, the macros `USING_NCBI_SCOPE` is used.

A number of macros have been defined to handle non-standard behavior of C++ compilers. For example, a macro `BREAK` is defined, that is used to break out of a loop, instead of using the break statement directly. This is done to handle a bug in the Sun WorkShop (pre 5.3 version) compiler that fails to call destructors for objects created in for-loop initializers. Another example is that some compilers (example, Sun Pro 4.2) do not understand the using namespace std; statement. Therefore, for portable code, the using namespace statement should be prohibited.

More details on the use of portable macros are presented in a later chapter.

Stream Wrappers

Not all C++ compilers support the templated iostreams, <iostream>, definitions; many support only the older non-templated <iostream.h> versions that place all definitions in a global name space. Also, some implementations of <iostream< are buggy. To support portable behavior the CORELIB defines the *ncbistre.hpp* file which triggers the proper inclusion of the appropriate iostream version. Unless, otherwise specified, the older version of *iostream.h*, if available, is used by default.

Once ncbistre.hpp has been included, the class definitions ***CNcbiIStream*** and ***CNcbiOStream*** are used instead of the standard istream and ostream. And, `NcbiCin` and `NcbiCout` are used instead of the standard `cin` and `cout`. ***CNcbiOstream*** and ***CNcbiIstream*** are defined as C-language typedefs. On Solaris and Windows, these are identical to the standard library output stream (ostream) and input stream (istream) classes. These typedefs are used on older computers to switch between the old stream library and the new standard library stream classes.

More details on the portable time class are presented in a later chapter.

String Manipulations

C++ defines the standard string class that provides operations on strings. However, compilers may exhibit non-portable string behavior especially with regards to multi-threaded programs. The CORELIB provides portable string manipulation facilities through the NStr class that provides a number of class-wide functions for string manipulation.

NStr portable functions include the string-to-X and X-to-string conversion functions where X is a data type including a pointer type, string comparsions with and without case, pattern searches within a string, string truncation, substring replacements, string splitting and join operations, string tokenization, etc.

Template Utilities

The C++ Template clases support a number of useful template classes for data structures such as vectors, lists. sets, maps, and so on.

The CORELIB defines a number of useful utility template classes. Some examples are template classes and functions for checking for equality of objects through a pointer, checking for non-null values of pointers, getting and setting map elements, deleting all elements from a container of pointers where the container can be a list, vector, set, multiset, map or multimap.

More details on the template utilities are presented in a later chapter.

Threads

Applications can run faster, if they are structured to exploit any inherent parallelism in the application's code execution paths. Code execution paths in an application can be assigned to separate threads. When the application is run on a multiprocessor system, there can be significant improvements in performance especially when threads run in parallel on separate processors.

The CORELIB defines a portable ***CThread*** class that can be used to provide basic thread functionality such as thread creation, thread execution, thread termination, and thread cleanup.

To create user defined threads you need to derive your class from ***CThread***, and override the thread's ***Main()*** method and, and if necessary the ***OnExit()*** method for thread-specific cleanup. Next, you create a thread object by instantiating the class you derived from ***CThread***. Now you are ready to launch thread execution by calling the thread's ***Run()*** method. The ***Run()*** method starts thread execution and the thread will continue to run until it terminates. If you want the thread to run independently of the parent thread you call the thread's ***Detach()*** method. If you want to wait till the thread terminates, you call the thread's ***Join()*** method.

Synchronization between threads is provided through mutexes and read/write locks.

More details on threads and synchronization is presented in a later chapter.

Time

The ***CTime*** class provides a portable interface to date and time functions. ***CTime*** can operate with both local and UTC time, and can be used to store data and time at a particular moment or elaspsed time. The time epoch is defined as Jan 1, 1900 so you cannot use ***CTime*** for storing timestamps before Jan 1, 1900.

The ***CTime*** class can adjust for daylight savings time. For display purposes, the time format can be set to a variety of time formats specified by a format string. For example, "M/D/Y h:m:s" for a timestamp of "5/6/03 14:07:09". Additional time format specifiers are defined for full month name (B), abbreviated month name (b), nanosecond (S), timezone fomat (Z), full weekday name (W) and abbreviated weekday name (w).

A class ***CStopWatch*** is also available that acts as a stop watch and measures elapsed time via the ***Elapsed()*** method, after its ***Start()*** method is called.

More details on threads and synchronization is presented in a later chapter.

The ALGORITHM Module

The ALGORITHM module is a collection of rigorously defined, often computationally intensive algorithms performed on sequences. It is divided into three groups:

- ALIGN. A set of global alignment algorithms, including generic Needleman-Wunsch, a linear-space Hirschberg's algorithm and a spliced (cDna/mRna-to-Genomic) alignment algorithm.
- BLAST. Basic Local Alignment Tool code and interface.
- SEQUENCE. Various algorithms on biological sequences, including antigenic determinant prediction, CPG-island finder, ORF finder, string matcher and others.

The CGI Module

The CGI module provides an integrated framework for writing CGI applications. It consists of classes that implement the CGI (Common Gateway Interface). These classes are used to retrieve and parse an HTTP request, and then compose and deliver an HTTP response.

The CGI module consists of a number of classes. The interaction between these classes is fairly complex, and therefore, not covered in this introductory chapter. We will attempt to only identify the major classes in this overview, and cover the details of their interaction in later chapters. Amongst the more important of the CGI classes are the ***CCgiApplication***, ***CCgiContext***, ***CCgiRequest***, ***CCgiResponse***, and ***CCgiCookie***.

The ***CCgiApplication*** is used to define the CGI application and is derived from the ***CNcbiApplication*** discussed eariler. You write a CGI application by deriving application class from ***CCgiApplication*** and providing an adaption of the Init(), Run(), and Exit() methods inherited from the ***CNcbiApplication*** class. Details on how to implement the Init(), Run() and Exit() methods for a CGI application are provided in a later chapter.

The ***CCgiRequest*** class is defined to receive and parse the request, and the ***CCgiResponse*** class outputs the response to an output stream.

The ***CCgiCookie*** class models a *cookie*. A cookie is a name, value string pair that can be stored on the user's web browser in an attempt to remember a session state. All incoming ***CCgiCookies*** are parsed and stored by the ***CCgiRequest*** object, and the outgoing cookies are sent along with the response by the ***CCgiResponse*** object.

The CGI application executes in a 'context' defined by the ***CCgiContext*** class. The ***CCgiContext*** class provides a wrapper for the ***CCgiApplication***, ***CCgiRequest*** and ***CCgiResponse*** objects and drives the processing of input requests.

More details on CGI classes and their interactions are presented in a later chapter.

The CONNECT Module

The CONNECT module implements a variety of interfaces and classes dealing with making connections to a network services. The core of the Connection Library is written in C which provides a low level interface to the communication protocols. The CONNECT module provides C++ interfaces so that the objects have diagnostic and error handling capabilities that are consistent with the rest of the toolkit. The standard sockets (SOCK) API is implemented on a variety of platforms such as Unix, MS-Windows, MacOS, Darwin. The CONNECT module provides a higher level access to the SOCK API by using C++ wrapper classes.

The following is a list of topics presented in this section:

- Socket classes
- Connector and Connection Handles
- Connection Streams
- Sendmail API
- Threaded Server

Socket classes

The C++ classes that implement the socket interface are ***CSocket***, ***CDatagramSocket***, ***CListeningSocket***, and ***CSocketAPI***. The socket defines an end point for a connection which consists of an IP address (or host name) of the end point, port number and transport protocol used (TCP, UDP).

The ***CSocket*** class encapsulates the descriptions of both local and remote end points. The local end point, which is the end point on the client issuing a connection request, is defined as parameters to the ***CSocket*** constructor. The remote end point on which the network service is running is specified as parameters to the ***Connect()*** method for the ***CSocket*** class. The ***CSocket*** class defines additional methods to manage the connection such as ***Reconnect()*** to reconnect to the same end point as the ***Connect()*** method; the ***Shutdown()*** method to terminate the connection; the ***Wait()*** method to wait on several sockets at once; the ***Read()*** and ***Write()*** methods to read and write data via the socket; and a number of other support methods.

The ***CSocket*** is designed for connection-oriented services such as those running over the TCP transport protocol. For connectionless, or datagram services, such as those running over the UDP transport protocol, you must use the ***CDatagramSocket*** class. The local end point is defined as parameters to the ***CDatagramSocket*** constructor. The remote end point is specified as parameters to the ***Connect()*** method for the ***CDatagramSocket*** class. Unlike the case of the connection-oriented services, this ***Connect()*** method only specifies the default destination address, and does not restrict the source address of the incoming messages. The methods ***Send()*** and ***Recv()*** are used to send the datagram, and the method ***SetBroadcast()*** sets the socket to broadcast messages sent to the datagram socket. The ***CDatagramSocket*** is derived from the ***CSocket*** class but methods such as ***Shutdown()*** and ***Reconnect()*** that apply to connection-oriented services are not available to users of the ***CDatagramSocket*** class.

The ***CListeningSocket*** is used by server-side applications to listen for connection requests. The ***CListeningSocket*** constructor specifies the port to listen to and the size of the connection request queue. You can change the port that the server application listens to any time by using the ***Listen()*** method. The ***Accept()*** method accepts the connection request, and returns a ***CSocket*** object through which data is transferred.

The ***CSocketAPI*** is a C++ wrapper for class-wide common socket utility functions available for sockets such as the ***gethostname()***, ***gethostbyaddr()***, ***ntoa()***, ***aton()***, and so on.

Connector and Connection Handles

The SOCK interface is a relatively low-level interface for connection services. The CONNECT module provides a generalization of this interface to connection services using a connection type and specialized connectors.

A connection is modeled by a connection type and a connector type. The connector type models the end point of the connection, and the connection type, the actual connection. Together, the connector and connection objects are used to define the following types of connections: socket, file, http, memory, and a general service connection.

The connector is described by a connector handle, CONNECTOR. CONNECTOR is a typedef and defined as a pointer to an internal data structure.

The connection is described by a connection handle CONN. CONN is a typedef and defined as a pointer to an internal structure. The CONN type is used as a parameter to a number of functions that handle the connection such as ***CONN_Create()***, ***CONN_ReInit()***, ***CONN_Read()***, ***CONN_Write()***, etc.

The CONNECTOR socket handle is created by a call to the ***SOCK_CreateConnector()*** function and passed the host name to connect to, the port number on the host to connect to, and maximum number of retries. The CONNECTOR handle is then passed as an argument to the ***CONN_Create()*** which returns a CONNECTION handle. The CONNECTION handle is then used with the connection functions (that have the prefix CONN_) to process the connection. The connection so created is bi-directional (full duplex) and input and output data can be processed simultaneously.

The other connector types, file, http, memory are similar to the socket connector type. In the case of a file connector, the connector handle is created by calling the ***FILE_CreateConnector()*** function and passed an input file and an output file. This connector could be used for both reading and writing files, when input comes from one file, and output goes to another file. This differs from normal file I/O when a single handle is used to access only one file, but resembles data exchange via sockets, instead. In the case of the HTTP connection, the HTTP_CreateConnector type is called and passed a pointer to network information structure, a pointer to a user-header consisting of HTTP tag-values, and a bitmask representing flags that affect the HTTP response.

The general service connector is the most complex connector in the library, and can model any type of service. It can be used for data transfer between an application and a named service. The data can be sent via HTTP or directly as a byte stream (using SOCK directly). In the former case it uses the HTTP connectors and in the latter the SOCK connectors. The general service connector is used when the other connector types are not adequate for implementing the task on hand.

More details on connector classes are presented in a later chapter.

Connection Streams

The CONNECT module provides a higher level of abstraction to connection programming in the form of C++ connection stream classes derived from the standard iostream class. This makes the familiar stream I/O operators, manipulators available to the connection stream. The main connection stream classes are the ***CConn_IOStream***, ***CCon_SocketStream***, ***CCon_HttpStream***, ***CCon_ServiceStream***, and ***CCon_MemoryStream***.

Figure 2 shows the relationship between the different stream classes. From this figure we can see that ***CConn_IOStream*** is derived from the C++ iostream class and serves as a base class for all the other connection stream classes. The ***CCon_IOStream*** allows input operations to be be tied to the output operations so that any input attempt first flushes the output queue from the internal buffers.

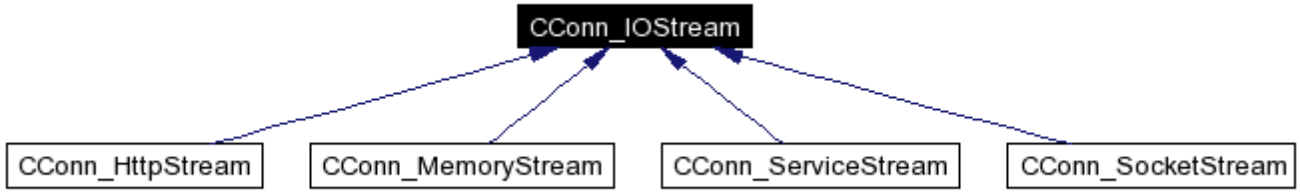

Figure 2: Connection stream classes

The ***CCon_SocketStream*** stream models a stream of bytes in a bi-directional TCP connection between two end points specified by a host/port pair. As the name suggests the socket stream uses the socket interface directly. The ***CCon_HttpStream*** stream models a stream of data between and HTTP client and an HTTP server (such as a web server). The server end of the stream is identified by a URL of the the form http://host[:port]/path[?args]. the ***CCon_ServiceStream*** stream models data transfer with a named service that can be found via dispatcher/load-

balancing daemon and implemented as either HTTP CGI, standalone server, or NCBI service. The ***CCon_MemoryStream*** stream models data transfer in memory similar to the C++ strstream class.

More details on connection stream classes are presented in a later chapter.

Sendmail API

The CONNECT module provides a number of APIs that provide access to sendmail. Sendmail is a popular MTA (Message Transfer Agent) found on many systems.

To initiate the use of sendmail. you must call the ***SendMailInfo_Int()*** function and pass it a properly initialized structure containing information such as that expected in a mail header (To, From, CC, BCC fields) and other communication settings. Then, you can send mail using the ***CORE_SendMail()*** or ***CORE_SendMailEx()*** functions.

Threaded Server

The CONNECT moduel provides support for multithreaded servers through the ***CThreaded-Server*** class. The ***CThreadedServer*** class is an abstract class for network servers and uses thread pools. This class maintains a pool of threads, called worker threads, to process incoming connections. Each connection gets assigned to one of the worker threads, allowing the server to handle multiple requests in parallel while still checking for new requests.

You must derive your threaded server from the ***CThreadedServer*** class and define the ***Process()*** method to indicate what to do with each incoming connection. The ***Process()*** method runs asynchronously by using a separate thread for each request.

More details on threaded server classes are presented in a later chapter.

The CTOOL Module

The CTOOL module provides bridge mechansims and conversion functions. More specifically, the CTOOL module provides a number of useful functions such as a bridge between the NCBI C++ Toolkit and the older C Toolkit for error handling, an ASN.1 connections stream that builds on top of the connection stream, and an ASN converter that provides templates for converting ASN.1-based objects between NCBI's C and C++ in-memory layouts.

The ASN.1 connections support is provides through functions ***CreateAsnConn()*** for creating an ASN stream connection; ***CreateAsnConn_ServiceEx()*** for creating a service connection using the service name, type and connection parameters; and ***CreateAsnConn_Service()*** which is a specialized case of ***CreateAsnConn_ServiceEx()*** with some parameters set to zero.

The DBAPI Module

The DBAPI module supports object oriented access to databases by providing user classes that model a database as a data source to which a connection can be made, and on which ordinary SQL queries or stored procedure SQL queries can be issued. The results obtained can be navigate using a result class or using the 'cursor' mechanism that is common to many databases.

The user classes are used by a programmer to access the database. The user classes depend upon a database driver to allow low level access to the underlying relational database management system (RDBMS). Each type of RDBMS can be expected to have a different driver that provides this low level hook into the database. The database drivers are architected to provide a uniform interface to the user classes so that the database driver can be changed to connect to a different database without affecting the program code that makes use of the user classes. For a list of the database drivers for different database that are supported, consult the Supported DBAPI Drivers section in a later chapter.

The following is a list of topics presented in this section:

- Database User Classes
- Database Driver Architecture

Database User Classes

The interface to the database is provided by a number of C++ classes such as the ***IDataSource, IDbConnection, IStatement, ICallableStatement, ICursor, IResultSet, IResultSetMetaData*** . The user does not use these interfaces directly. Instead, the DBAPI module provides concrete classes that implement these interface classes. The corresponding concrete classes for the above mentioned interfaces are ***CDataSource, CDbConnection, CStatement, CCallableStatement, CCursor, CResultSet, CResultSetMetaData***.

Before accessing to a specific database, the user must register the driver with the ***CDriverManager*** class which maintains the drivers registered for the application. The user does this by using the ***CDriverManager*** class' factory method ***GetInstance()*** to create an instance of the ***CDriverManager*** class and registering the driver with this driver manager object. For details on how this can be done, consult the section Choosing the Driver in a later chapter.

After the driver has been registered, the user classes can be used to access that database. There are a number of ways this can be done, but the most common method is to call the ***IDataSource*** factory method ***CreateDs()*** to create an instance of the data source. Next, the ***CreateConnection()*** method for the data source is called, to return a connection object that implements the ***IConnection*** interface. Next, the connection object's ***Connect()*** method is called with the user name, password, server name, database name to make the connection to the database. Next, the connection object's ***CreateStatement()*** method is called to create a statement object that implements the ***IStatement*** interface. Next, the statement object's ***Execute()*** method is called to execute the query. Note that additional calls to the ***IConnection::CreateStatement()*** results in cloning the connection for each statement which means that these connections inherit the database which was specified in the ***Connect()*** or ***SetDatabase()*** method.

Executing the statement objects's ***Execute()*** method returns the result set which is stored in the statement object and can be accessed using the statement object's ***GetResultSet()*** method. You can then call the statement object's ***HasRows()*** method which returns a boolean true if there are rows to be processed. The type of the result can be obtained by calling the ***IResultSet::***

GetResultType() method. The ***IStatement::ExecuteUpdate()*** method is used for SQL statements that do not return rows (UPDATE or DELETE SQL statement), in which case the method ***IStatement::GetRowCount()*** returns the number of updated or deleted rows.

Calling the ***IStatement::GetResultSet()*** returns the rows via the result set object that implements the IResultSet interface. The method ***IResultSet::Next()*** is used to fetch each row in the result set and retruns a false when no more fetch data is available; otherwise, it returns a true. All column data, except BLOB data is represented by a ***CVariant*** object. The method ***IResultSet::GetVariant()*** takes the column number as its parameter where the first column has the start value of 1.

The ***CVariant*** class is used to describe the fields of a record which can be of any data type. The ***CVariant*** has a set of accessor methods (GetXXX()) to extract a value of a particular type. For example, the ***GetInt4(), GetByte(), GetString()***, methods will extract an Int4, Byte data value from the ***CVariant*** object. If data extraction is not possible because of incompatible types, the ***CVariantException*** is thrown. The ***CVariant*** has a set of factory methods for creating objects of a particular data type, such as ***CVariant::BigInt()*** for Int8, ***CVariant::SmallDateTime()*** for NCBI's ***CTime***, and so on.

For details on sample code illustrating the above mentioned steps consult the sections Data Source and Connections and Main Loop in a later chapter.

Database Driver Architecture

The driver can use two different methods to access the particular RDBMS. If RDBMS provides a client library (CTLib) for a given computer system, then the driver utilizes this library. If there is no client library, then the driver connects to RDBMS through a special gateway server which is running on a computer system where such library does exist.

the database driver architecture has two major groups of the driver's objects: the RDBMS independent objects, and the RDBMS dependent objects specific to a RDBMS. From a user's perspective, the most important RDBMS dependent object is the driver context object. A connection to the database is made by calling the driver context's ***Connect()*** method. All driver contexts implement the same interface defined in the ***I_DriverContext*** class.

If the application needs to connect to RDBMS libraries from different vendors, there is a problem trying to link statically with the RDBMS libraries from different vendors. The reason for this is that most of these libraries are written in C, and may use the same names which causes name collisons. Therefore, the ***C_DriverMgr*** is used to overcome this problem and allow the creation of a mixture of statically linked and dynamically loaded drivers and use them together in one executable.

The low level connection to an RDBMS is specific to that RDBMS. To provide RDBMS independence, the connection information is wrapped in an RDBMS independent object ***CDB_Connection***. The commands and the results are also wrapped in an RDBMS independent object. The user is responsible for deleting these RDBMS independent objects because the life spans of the RDBMS dependent and RDBMS independent objects are not necessarily the same.

Once you have the ***CDB_Connection*** object, you can use it as a factory for the different types of command objects. The command object's ***Result()*** method can be called to get the results. To send and to receive the data through the driver you musr use the driver provided datatypes such as ***CDB_BigInt, CDB_Float, CDB_SmallDateTime***. These driver data types are all derived from ***CDB_Object*** class.

More details on the database driver architecture is presented in a later chapter.

The GUI Module

The GUI Module has been designed for scientific visualization of biological sequences. The GUI SEQ library describes and implements a set of objects needed to display and navigate molecule sequences and features. The basic functionality allows the display a molecule sequence and features, use mouse or keyboard to select parts of the sequence, get feature information, change features shape, change various interface colors.

An advantage of using the SeqView is that you can have multiple sequence data sources and can quickly and easily switch between them (see Figure 1 of the GUI chapter).

The Sequence View presented in Figure 1 of the GUI chapter relies on OpenGL, a widely used graphical library for interactive 2D and 3D graphics applications; and FLTK a cross-platform C++ GUI toolkit. FLTK is used to layout and display GUI elements. The graphical component layout for the Sequence View can be quickly done using FLUID which is FLTK's UI (User Interface) builder.

Three major classes are used to build the Sequence View. These are the ***CSeqView***, ***CSeqPanel***, and the ***CSeqDataSource***. The ***CSeqView*** represents the displayed Sequence View itself and consists of a ***CSeqPanel*** and FLTK's Fl_Scrollbar object. The ***CSeqPanel*** represents the panel graphical element inherited from FLTK's Fl_Gl_Window class. The ***CSeqDataSource*** represents the bio sequence data source that is displayed. User defined sequence views are created by deriving from the ***CSeqView*** class which also is used to handle all mouse and keyboard events.

In order to set up a Sequence View, you have to create an instance of ***CSeqView***, define your data source by inheriting from ***CSeqDataSource*** and implementing the required methods, and registering the data source with Sequence View.

More details on the GUI module is presented in a later chapter.

The HTML Module

The HTML module implements a number of HTML classes that are intended for use in CGI and other programs. The HTML classes can be used to generate HTML code dynamically.

The HTML classes can be used to parse an HTML page and represent it internally in memory as a graph. Each HTML element or tag is represented by a node in the graph. The attributes for an HTML element are represented as attributes in the node. A node in the graph can have other elements as children. For example, for an HTML page, the top HTML element will be described by an HTML node in the graph. The HTML node will have the HEAD and BODY nodes as its chil-

dren. The BODY node will have text data and other HTML nodes as its children. The graph structure representation of an HTML page allows easy additions, deletions and modification of the page elements.

The following is a list of topics presented in this section:

- Relationships between HTML classes
- HTML Processing

Relationships between HTML classes

The base class for all nodes in the graph structure for an HTML document is the ***CNCBINode***. The ***CNCBINode*** class is derived from ***CObject*** and provides the ability to add, delete, modify the nodes in the graph. The ability to add and modify nodes is inherited by all the classes derived from ***CNCBINode*** (see Figure 3). The classes derived from ***CNCBINode*** represent the HTML elements on an HTML page. You can easily identify the HTML element that a class handles from the class names such as ***CHTMLText***, ***CHTMLButtonList***, etc.

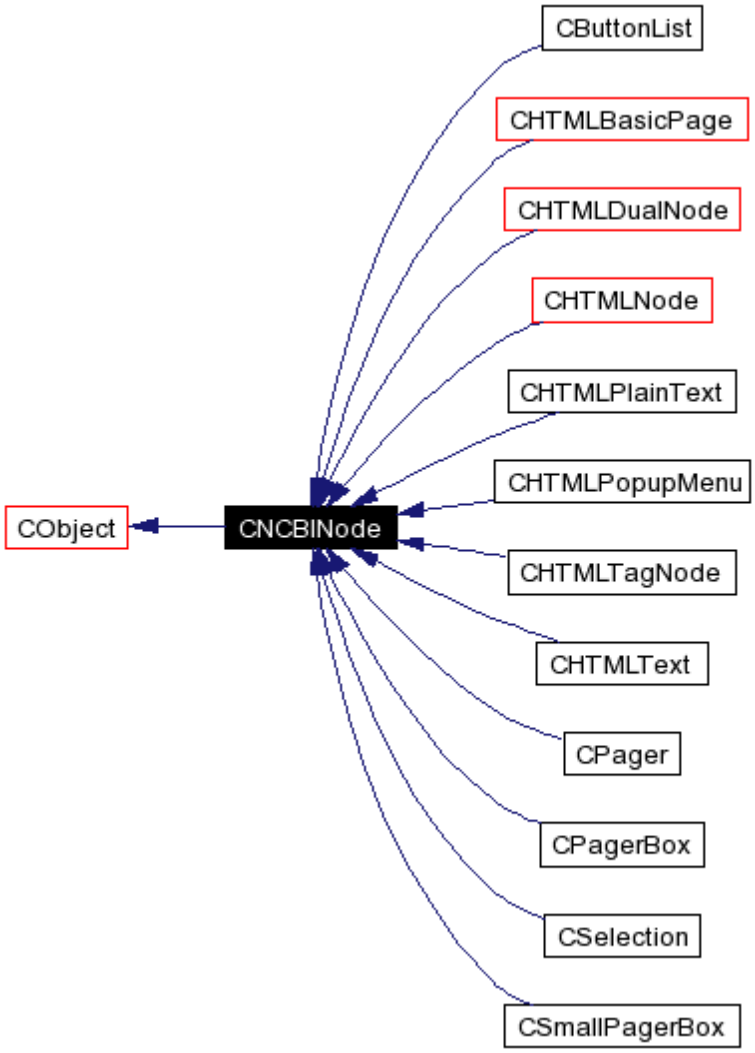

Figure 3: HTML classes derived from ***CNCBINode***

The text node classes ***CHTMLText*** and ***CHTMLPlainText*** are intended to be used directly by the user. Both ***CHTMLText*** and ***CHTMLPlainText*** are used to insert text into the generated html, with the difference that ***CHTMLPlainText*** class performs HTML encoding before generation. A number of other classes such as ***CHTMLNode***, ***CHTMLElement***, ***CHTMLOpenElement***, and ***CHTMLListElement*** are base classes for the elements actually used to construct an HTML page, such as ***CHTML_head***, ***CHTML_form*** (see Figure 4).

The ***CHTMLNode*** class is the base class for ***CHTMLElement*** and ***CHTMLOpenElement*** and is used for describing the HTML elements that are found in an HTML page such as HEAD, BODY, H1, BR, etc. The ***CHTMLElement*** tag describes those tags that have a close tag and are well formed. The ***CHTMLOpenElement*** class describes tags that are often found without the corresponding close tag such as the BR element that inserts a line break. The ***CHTMLListElement*** class is used in lists such as the OL element.

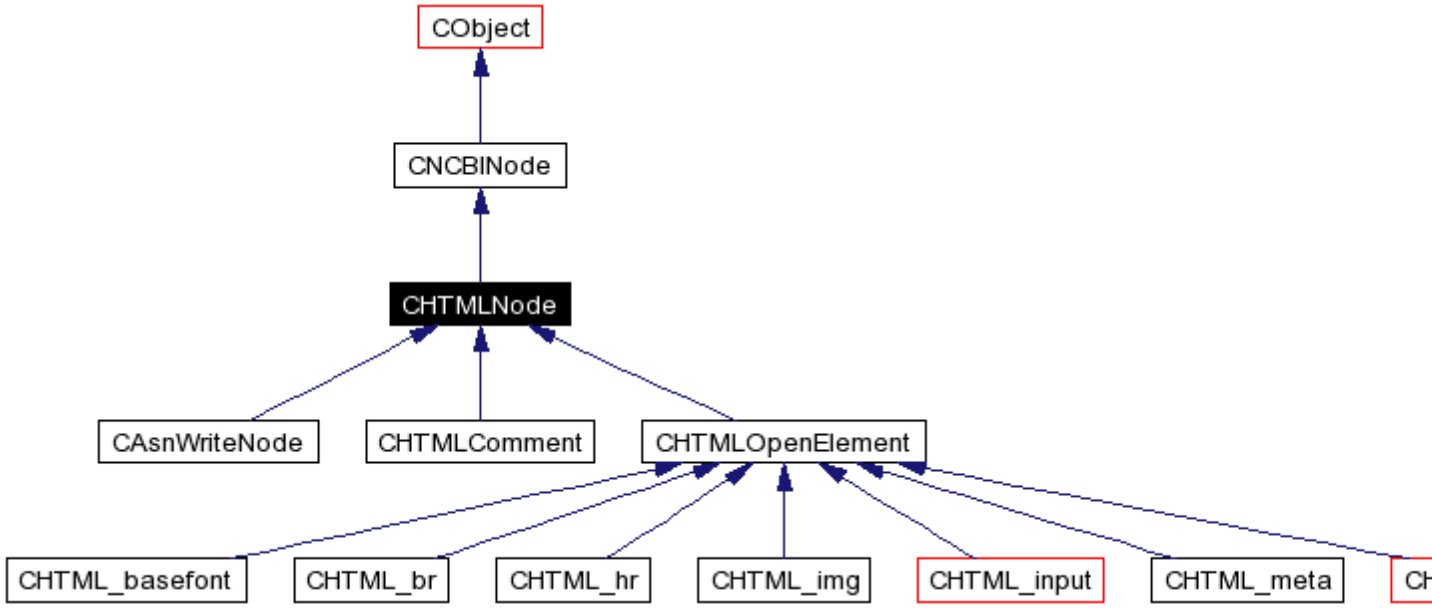

Figure 4: The ***CHTMLNode*** class and its derived classes

An important class of HTML elements used in forms to input data are the input elements such as checkboxes, radio buttons, text fields, etc. The ***CHTML_input*** class derived from the ***CHTML_OpenElement*** class serves as the base class for a variety of input elemens (see Figure 5)

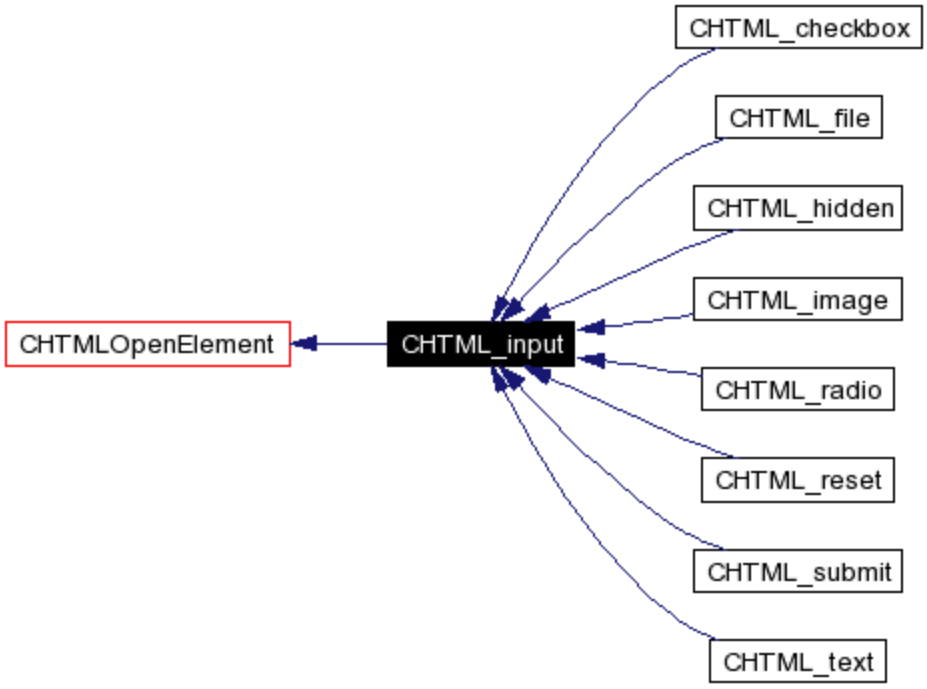

Figure 5: The ***CHTML_input*** class and its derived classes

More details on HTML classes and their relationships is presented in a later chapter.

HTML Processing

The HTML classes can be used to dynamically generate pages. In addition to the classes described in the previous section, there are a number of page classes that are designed to help with HTML processing. The page classes serve as generalized containers for collections of other HTML components, which are mapped to the page. Figure 6 describes the important classes in page class hierarchy.

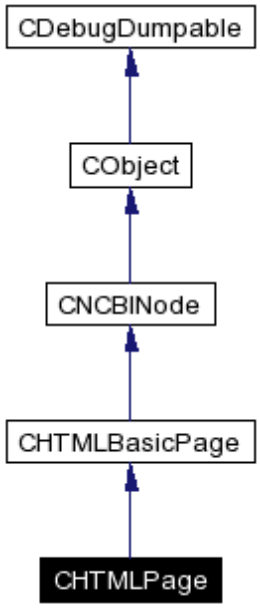

Figure 6: HTML page classes

the ***CHTMLBasicPage*** class is as a base class whose features are inherited by the ***CHTML-Page*** derived class - it is not intended for direct usage. Through the methods of this class, you can access or set the CgiApplication, Style, and TagMap stored in the class.

The ***CHTMLPage*** class when used with the appropriate HTML template file, can generate the 'bolier plate' web pages such as a standard corporate web page, with a corporate logo, a hook for an application-specific logo, a top menubar of links to several databases served by a query program, a links sidebar for application-specific links to relevant sites, a VIEW tag for an application's web interface, a bottom menubar for help links, disclaimers, and other boiler plate links. The template file is a simple HTML text file with named tags (<@tagname@>) which allow the insertion of new HTML blocks into a pre-formatted page.

More details on ***CHTMLBasicPage***, ***CHTMLPage*** and related classes is presented in a later chapter.

The OBJECT MANAGER Module

The Object Manager module is a library of C++ classes, which facilitate access to biological sequence data. It makes it possible to transparently download data from the GenBank database, investigate biological sequence data structure, retrieve sequence data, descriptions and annotations.

The Object Manager has been designed to present an interface to users that minimizes their exposure to the details of interacting with biological databases and their underlying data structures. The Object Manager, therefore, coordinates the use of biological sequence data objects, particularly the management of the details of loading data from different data sources.

The NCBI databases and software tools are designed around a particular model of biological sequence data. The data model must be very flexible because the nature of this data is not yet fully understood, and its fundamental properties and relationships are constantly being revised. NCBI uses Abstract Syntax Notation One (ASN.1) as a formal language to describe biological sequence data and its associated information.

The bio sequence data may be huge and downloading all of this data may not be practical or desirable. Therefore, the Object Manager transparently transmits only the data that is really needed, and not all of it at once. There is a datatool that generates corresponding data objects (source code and header files) from the object's ASN.1 specification. The Object Manager is able to manipulate these objects.

Biological sequences are identified by a Seq_id, which may have different forms.

The most general container object of bio sequence data, as defined in NCBI data model, is Seq_entry. A great deal of NCBI software is designed to accept a Seq_entry as the primary unit of data. In general, the Seq_entry is defined recursively as a tree of Seq_entry objects, where each node contains either Bioseq or list of other Seq_entry objects and additional data like sequence description, sequence annotations.

Two important concepts in the Object Manager are *scope* and *reference resolution*. The client defines a scope as the sources of data where the system uses only "allowed" sources to look for data. Scopes may contain several variants of the same bio sequence (Seq_entry). Since sequences refer to each other, the scope sets may have some data that is common to both scopes. In this case changing data in one scope should be reflected in all other scopes, which "look" at the same data.

The other concept a client uses is *reference resolution*. Reference resolution is used in situations where different biological sequences can refer to each other. For example, a sequence of amino acids may be the same as sequence of amino acids in an another sequence. The data retrieval system should be able to resolve such references automatically answering what amino acids are actually here. Optionally, at the client's request, such automatic resolution may be turned off.

The Object Manager provides a consistent view of the data despite modifications to the data. For example, the data may change during a client's session because new biological data has been uploaded to the database while the client is still processing the old data. In this case, when the client for additional data, the system should retrieve the original bio sequence data, and not

the most recent one. However, if the database changes between a client's sessions, then the next time the client session is started, the most recent data is retrieved, unless the client specifically asks for the older data.

The Object Manager is thread safe, and supports multithreading which makes it possible to work with bio sequence data from multiple threads.

The Object Manager includes numerous classes for accessing bio sequence data such as ***CDataLoader*** and ***CDataSource*** which manage global and local accesses to data, ***CSeqVector*** and ***CSeqMap*** objects to find and manipulate sequence data, a number of specialized iterators to parse descriptions and annotations, among others. The ***CObjectManager*** and ***CScope*** classes provide the foundation of the library, managing data objects and coordinating their interactions.

More details on the Object Manager and related classes is presented in a later chapter.

The SERIAL Module

The SERIAL Module provides serialization support for objects. Serialization provides persistence of objects. Normally objects are transient and have an existence only when they are resident in memory. When the application that controls these objects terminates, these objects are released from memory. Using serialization, the objects in memory can be written out to an output stream such as a file, and read back again when the program restarts. This type of application of serialization is called Lightweight persistence and is used for the archival of objects to be used in a later invocation of the same program. Another example of serialization is to pass objects between independent processes. The processes could be running on different machines in which case the objects could be serialized and passed via sockets using a sockets stream. This type of serialization is used in Remote Method Invocation (RMI) which requires objects to be passed between processes.

Serializaing objects requires reading of an object from an input stream and writing of an object to an output stream. The data that is read and written describes the object's internal structure and specifies an encoding scheme to encode the data. The input and output streams are implemented as a set of object stream classes. The objects are described by using the classes defined in the Object Manager module.

The base classes for the object stream classes are ***CObjectIStream*** and ***CObjectOStream***. All the specialized object stream format are handled by subclassing these base classes. Examples of these specialized formats including XML, binary ASN.1, and text ASN.1.

The SERIAL module implements specialized read (>>) and write (<<) operator methods for these classes, so when the read or write operator method is used on an object, that object is read from or written to the specified object stream.

The datatool is an important tool built using the SERIAL module classes. The datatool can be used to generate C++ data storage classes based on the ASN.1 or DTD specifications. Additionally, the datatool can be used to convert the ASN.1 specification into DTD and vice versa, and convert data between ASN.1 and XML formats. For more information on the datatool please the later chapter on tools.

The following sections describe the more important classes in the SERIAL module:

- The Input Stream Class
- The Output Stream Class
- The Hook Classes

The Input Stream Class

As mentioned earlier, the ***CObjectIStream*** and ***CObjectOStream*** serve as important virtual base classes for specialized object streams (see Figure X). These classes provide important method interfaces for the specialized object stream classes ***CObjectIStreamXML*** and ***CObjectOStreamXML*** for XML, ***CObjectIStreamAsnBinary*** and ***CObjectOStreamAsnBinary*** for binary ASN.1, ***CObjectIStreamAsn*** and ***CObjectOStreamAsn*** for text ASN.1.

The methods whose interfaces are described in the ***CObjectIStream*** class include ***Open()***, ***Close()***, ***GetDataFormat()***, ***Read()***, ***ReadObject()***, ***ReadSeparateObject()***, ***Skip()***, ***SkipObject()***, ***GetDataHeader()***, and ***ReadFileHeader()***.

The meanings of these methods are obvious from their names, but perhaps some methods require some explanation. The ***GetDataFormat()*** returns the enumerated ***ESerialDataFormat*** type for the stream. This enumerated type has values specific to each of the specialized formats. For example, enumeration constants eSerial_XML for XML format, eSerial_AsnText for ASN.1 text format, and eSerial_AsnBinary for ASN.1 binary format. The ***ReadFileHeader()*** method reads the first line from the file, and returns it in a string which can be used to compare against string constants such as "Seq-entry" , "Bioseq-entry" to determine the nature of objects stored in the input stream. In fact, the ***ReadFileHeader()*** method is called by the ***Read()*** method to determine the nature of objects in the stream.

The default behavior of ***Read()*** is to load the top-level object, along with all of its contained subobjects into memory. In some cases this may use up considerable memory, and it may be only the top-level object which is needed by the application. In this case, the method ***ReadObject()*** can be used to load subobjects as persistent data members of the root object, and the method ***ReadSeparateObject()*** can be used to load subobjects as temporary local objects.

The ***Skip()*** and ***SkipObject()*** methods allow entire top-level objects and subobjects to be "skipped" while data is being read. The input is still read from the stream and validated, but no object representation for that data is generated for the skipped objects. Instead, the data is stored in a delay buffer associated with the object input stream, where it can be accessed as needed. The ***Skip()*** should only be applied to top-level objects. The ***SkipObject()*** method may also be used to skip subobjects of the root object.

The user can also install type-specific read, write, and copy hooks, which can change the default behavior of loading objects where these hooks call the ***ReadSeparateObject()*** and ***SkipObject()*** methods where needed.

More details on ***CObjectIStream*** and related classes is presented in a later chapter.

The Output Stream Class

The output object stream classes complement the behavior of the ***CObjectIStream*** classes. The ***CObjectOStream*** virtual base class is used to derive the ***CObjectOStreamXml***, ***CObjectOStreamAsn***, and ***CObjectOStreamAsnBinary*** classes.

The methods whose interfaces are described in the***CObjectOStream*** class include ***Open()***, ***Close()***, ***GetDataFormat()***, ***Write()***, ***WriteObject()***, ***WriteSeparateObject()***, ***Flush()***, ***FlushBuffer()***, and ***WriteFileHeader()***.

The meanings of these methods are obvious from their names, but perhaps some methods require some explanation. The ***GetDataFormat()*** returns the enumerated ***ESerialDataFormat*** type for the stream. This enumerated type has values specific to each of the specialized formats. For example, enumeration constants *eSerial_XML* for XML format, *eSerial_AsnText* for ASN.1 text format, and *eSerial_AsnBinary* for ASN.1 binary format. The ***WriteFileHeader()*** method writes the first header line into the file that describes the file contents.This is the same line read by the ***ReadFileHeader()*** method for the inputr stream classes derived from ***CObjectIStream***.

The ***Write*()*** methods ccomplement the ***Read*()*** methods defined for the input streams. The ***Write()*** first calls ***WriteFileHeader()***, and then calls ***WriteObject()***. The ***WriteSeparateObject()*** can be used to write a temporary object and all its children objects as well, to the output stream. It is also possible to install type-specific *write* hooks that serve as wrapper functions that define what occurs immediately before and after the data is actually written.

More details on ***CObjectOStream*** and related classes is presented in a later chapter.

The Hook Classes

Since objects that need to be serialized can be very application-speciifc, the SERIAL module provides hook mechanisms, whereby the needed application-specific behavior can be installed in the object's static class, ***CTypeInfo***. These hooks can be installed **globally**, where they will be applied on **all** streams where these events occur, or **locally**, where they will only be applied to a selected stream.

For any given object and specific stream, at most one read hook and one write hook is "active". If let's say for instance, an application-specific object `myObject`, has a locally installed read hook as well as a global read hook, then the locally installed hook will override the global hook when a read occurs on the "local" stream. Read events on all of the other "non-local" streams will trigger the globally installed hook. You can have multiple read/write local and global hooks for a selected object. Older or less specific hooks are overridden by the more specific or most recently installed hooks.

The SERIAL module provides abstract base classes that are subclassed for creating new hooks. For read hooks these are the ***CReadObjectHook***, ***CReadClassMemberHook***, and the ***CReadChoiceVariantHook***. These classes are defined for the different contexts in which an object might be encountered on an input stream. For Write hooks these are the ***CWriteObjectHook*** class, the ***CWriteClassMemberHook*** class, and the ***CWriteChoiceVariantHook***.

These different contexts are when an object is encountered as a stand-alone object, as a data member of a containing object, or as a variant of a *choice* object

The UTIL Module

The UTIL module is collection of some very useful utility classes that implement I/O related functions, algorithms, container classes, text related and thread related functions. Individual facilities include classes to compute checksums, implement interval search trees, lightweight strings, string search, linked sets, random number generation, UTF-8 conversions, registry based DNS, rotating log streams, thread pools, and many others.

The following sections give an overview of the utility classes:

- Checksum
- Console Debug Dump Viewer
- Lightweight Strings
- Range Support
- Weak Map Templates
- Linked Sets
- Random Number Generator
- Registry based DNS
- Resizing Iterator
- Rotating Log Streams
- Stream Support
- String Search
- Thread Pools
- UTF 8 Conversion

Checksum

The Checksum class implements CRC32 (Cyclic Redundancy Checksum 32-bit) calculation. The CRC32 is a 32-bit polynomial checksum that has many applications such as verifying the integrity of a piece of data. The ***CChecksum*** class implements the CRC32 checksum that can be used to compute the CRC of a sequence of byte values.

The checksum calculation is set up by creating a ***CChecksum*** object using the ***CChecksum*** constructor and passing it the type of CRC to be calculated. Currently only CRC32 is defined, so you must pass it the enumeration constant *eCRC32* also defined in the classs.

Data on which the checksum is to be computed is passed to the ***CChecksum'sAddLine()*** or ***AddChars()*** method as a character array. As data is passed to these methods, the CRC is computed and stored in the class. You can get the value of the computed CRC using the ***GetChecksum()*** method. Alternatively, you can use the ***WriteChecksum()*** method and pass it a ***CNcbiOstream*** object and have the CRC written to the output stream in the folllowing syntax:

/* Original file checksum: lines: *nnnn*, chars: *nnnn*, CRC32: *xxxxxxxx* */

Console Debug Dump Viewer

The UTIL module implements a simple Console Debug Dump Viewer that enables the printing of object information on the console, through a simple console interface. Objects that can be debugged must be inherited from ***CDebugDumpable*** class. The ***CObject*** is derived from ***CDebugDumpable***, and since most other objects are derived from ***CObject*** this makes these objects 'debuggable'.

The Console Debug Dump Viewer is implemented by the ***CDebugDumpViewer*** class. This class implements a breakpoint method called ***Bpt()***. This method is called with the name of the object and a pointer to the object to be debugged. This method prompts the user for commands that the user can type from the console:

```
Console Debug Dump Viewer
Stopped at  testfile.cpp(120)
current object: myobj = xxxxxx
Available commands:
   t[ypeid]  address
   d[ump]    address  depth
   go
```

The ***CDebugDumpViewer*** class also permits the enabling and disabling of debug dump breakpoints from the registry.

Lightweight Strings

If you don't need the full functionality of the C++ string class, and want a lighter, and therefore, more efficient version of a string class, you can use the ***CLightString*** class that can be used to store character strings. Unlike the standard C++ string class, ***CLightString*** does not take ownership of the string. So, the string data should exist for the duration of holding the ***CLightString*** object. There is no explict destructor for the ***CLightString***class, so the string data should be deleted explicitly, if needed, after ***CLightString*** object destruction.

Another difference between the ***CLightString*** class and the standard C++ string class is that ***CLightString*** compares first by string length, and then by string contents. Therefore, the string "az" will sort before "abc", whereas in the standard lexicographial sort, "abc" will sort before "az".

More details on the ***CLightString*** class is presented in a later chapter.

Range Support

The UTIL module provides a number of container classes that support a *range* which models an interval consisting of a set of ordered values. the ***CRange*** class stores information about an interval, **[*from*, *to*]**, where the ***from*** and ***to*** points are inclusive. This is sometimes called a *closed interval*.

Another class, the ***CRangeMap*** class, is similar to the ***CRange*** class but allows for the storing and retrieving of data using the interval as key. The time for iterating over the interval is proportional to the amount of intervals produced by the iterator and may not be efficient in some cases.

Another class, the ***CIntervalTree*** class, has the same functionality as the ***CRangeMap*** class but uses a different algorithm; that is, one based on McCreight's algorithm. Unlike the ***CRangeMap*** class, the ***CIntervalTree*** class allows several values to have the same key interval. This class is faster and its speed is not affected by the type of data but it uses more memory (about three times as much as ***CRangeMap***) and, as a result, is less efficient when the amount of interval in the set is quite big. For example, the ***CIntervalTree*** class becomes less efficient than ***CRangeMap*** when the total memory becomes greater than processor cache.

More details on range classes is presented in a later chapter.

Weak Map Templates

The UTIL module provides two template classes, the ***CWeakMap*** class and the ***CWeakMapKey*** class that provide an extension to the standard C++ STL map class with the additional feature that it automatically removes elements from the map when corresponding key is 'destructed'.

The key is of type ***CWeakMapKey<Object>*** where the template parameter ***Object*** is typically a string type, for character value keys.

More details on Weak Map classes is presented in a later chapter.

Linked Sets

The UTIL module defines a template container class, ***CLinkedMultiset***, that can hold a a linked list of multiset container types.

The ***CLinkedMultiset*** defines iterator methods ***begin()***, ***end()***, ***find()***, ***lower_bound()***, ***upper_bound()***, to help traverse the container. The method ***get()***, fetches the contained value, the method ***insert()*** inserts a new value into the container, and the method ***erase()***, removes the specified value from the container.

Random Number Generator

The UTIL module defines the ***CRandom*** class that can be used for generating 32-bit unsigned random numbers. The random number generator algorithm is the Lagged Fibonacci Generator (LFG) algorithm.

The random number generator is initialized with a seed value, and then the ***GetRandom()*** method is called to get the next random number. You can also specify that the random number value that is returned be in a speciifed range of values.

Registry based DNS

The UTIL module defines the ***CSmallDns*** class that implements a simple registry based DNS server. The ***CSmallDns*** class provides DNS name to IP address translations similar to a standard DNS server, except that the database used to store DNS name to IP address mappings is a non-standard local database. The database of DNS names and IP address mappings are kept in a registry-like file named by local_hosts_file using section [LOCAL_DNS].

The ***CSmallDns*** has two methods that are responsible for providing the DNS name to IP address translations: the LocalResolveDNS method and the LocalBackResolveDNS method. The LocalResolveDNS method does 'forward' name resolution. That is, given a host name, it returns a string containing the IP address in the dotted decimal notation. The LocalBackResolveDNS method does a 'reverse lookup'. That is, given an IP address as a dotted decimal notation string, it returns the host name stored in the registry.

Resizing Iterator

The UTIL module defines two template classes, the ***CResizingIterator*** and the ***CConstResizingIterator*** classes that handle sequences represented as packed sequences of elements of different sizes For example, a vector <char> might actually hold 2-bit values, such as nucleotides, or 32-bit integer values.

The purpose of these iterator clases is to provide iterator semantics for data values that can be efficiently represented as a packed sequence of elements regardless of the size.

Rotating Log Streams

The UTIL module defines the ***CRotatingLogStream*** class that can be used to implement a rotating log file. The idea being that once the log of messages gets too large, a 'rotation' operation can be performed. The default rotation is to rename the existing log file by appending it with a timestamp, and opening a new log.

The rotating log can be specified as a file, with an upper limit (in bytes) to how big the log will grow. The ***CRotatingLogStream*** defines a method called ***Rotate()*** that implements the default rotation.

Stream Support

The UTIL module defines a number of portable classes that provide additional stream support beyond that provided by the standard C++ streams. The ***CByteSource*** class acts as an abstract base class (see Figure 7), for a number of stream classes derived from it. As the name of the other classes derived from ***CByteSource*** suggests, each of these classes provides the methods from reading from the named source. To list a few examples: ***CFileByteSource*** is a specialized class for reading from a named file; ***CMemoryByteSource*** is a specialized class for reading from a memory buffer; ***CResultByteSource*** is a specialized class for reading database results; ***CStreamByteSource*** is a specialized class from reading from the C++ input stream(istream); ***CFStreamByteSource*** is a specialized class from reading from the C++ input file stream (ifstream).

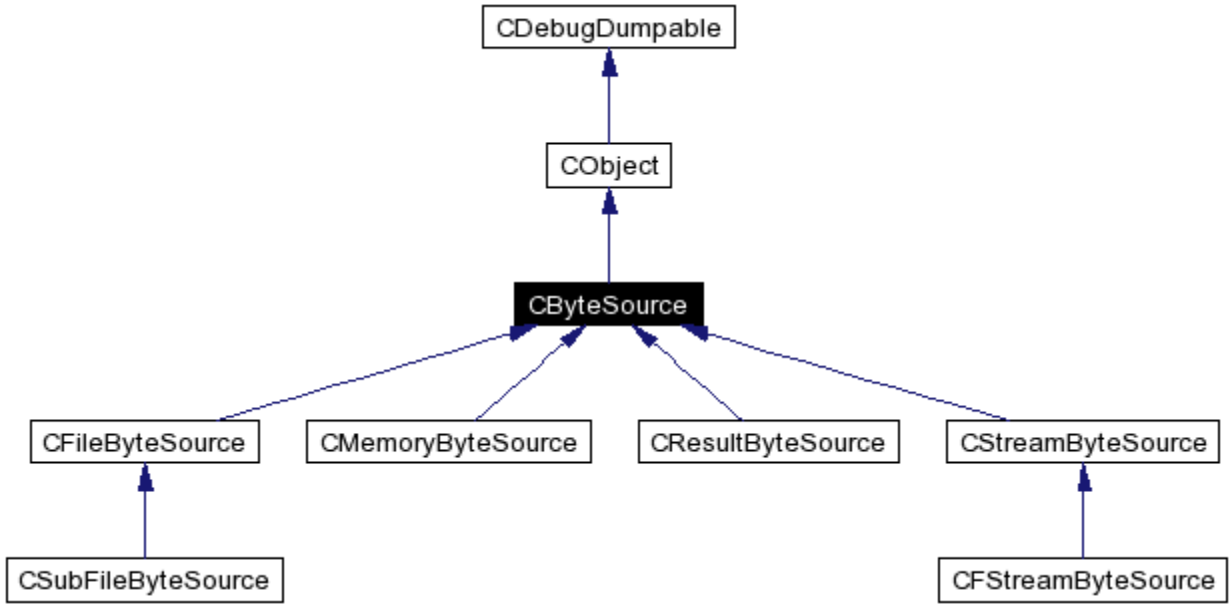

Figure 7: Relationship between ***CByteSource*** and its derived classes

The classes such as ***CSubFileByteSource*** are used to define a slice of the source stream in terms of a start position and a length. The read operations are then confined to this slice.

Additional classes, the ***CIStreamBuffer*** and the ***COStreamBuffer*** have been defined for standard input and output buffer streams. These can be used in situations where a compiler's implementation of the standard input and output stream buffering is inefficient.

More details on the stream classes are presented in a later chapter.

String Search

The UTIL module defines the ***CBoyerMooreMatcher*** class and the ***CFsmText*** class which are used for searching for a single pattern over varying texts.

The ***CBoyerMooreMatcher*** class, as the name suggests, uses the Boyer-Moore algorithm for string searches. The ***CFsmText*** is a template class that perfoms the search using a finite state automaton for a specified to be matched data type. Since the matched data type is often a string, the ***CTextFsa*** class is defined as a convenience by instantiating the ***CFsmText*** with the matched type template parameter set to string.

The search can be setup as a case sensitive or case insensitive search. The default is case sensitive search. In the case of the ***CBoyerMooreMatcher*** class, the search can be setup for any pattern match or a whole word match. A whole word match means that a pattern was found to be between white spaces. The default is any pattern match.

Thread Pools

The UTIL module defines a number of container classes to implement a pool of threads.

The ***CPoolOfThreads*** is a template abstract class that defines an interface for a pool of request-handling threads. The template parameter is for an arbitrary request type. The pool of threads is defined by a maximum size, the number of threads in the pool, and a threshold parameter that creates another thread automatically when the difference between unfinished requests and the number of threads in the pool exceeds this threshold.

The ***CStdRequest*** is an abstract class, derived from ***CObject***, that is used to model requests to the thread pool. The classes' interface method ***Process()*** is called when a thread handles the request. The ***CStdRequest*** is not directly used in the thread classes, but through a smart pointer implemented by CRef; that is, ***CRef<CStdRequest>***

From the above two class definitions, a new class is derived to model a pool of threads, the ***CStdPoolOfThreads***. The class ***CStdPoolOfThreads*** is derived from the template class ***CPoolOfThreads***by using ***CRef<CStdRequest>*** as the template parameter. Figure 8 shows the relationship between these classes.

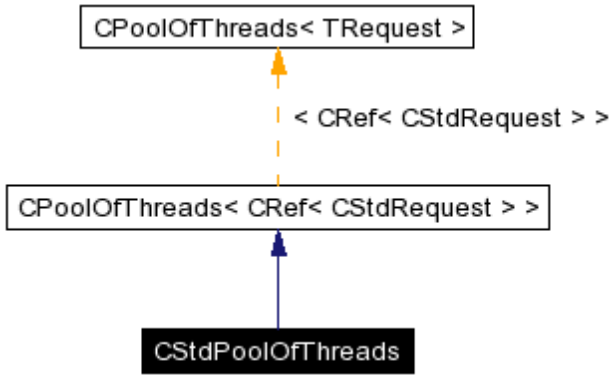

Figure 8: Relationship between pool of threads classes

The previously mentioned classes define a container or pool of threads. The actual threads which are placed inside the pool are derived from the ***CThread*** class. The template class ***CThreadInPool*** is an abstract class, derived from ***CThread***, and models the threads in a pool. The template parameter is for an arbitrary request type. The ***CStdThreadInPool*** is derived from the ***CThreadInPool*** by using ***CRef<CStdRequest>*** as the template parameter.

To summarize, the concrete classes are ***CStdPoolOfThreads*** to model the pool of threads, the ***CStdRequest*** to model the request, and the ***CStdThreadInPool*** to model the thread that is placed in the pool.

The thread classes use the ***CBlockingQueue*** class internally to manage requests. The ***CBlockingQueue*** class implements a blocking first-in-first-out queue container. The blocking nature of this container class ensures that an attempt to extract an element from an empty queue blocks efficiently until more elements are available.

More details on threaded pool classes is presented in a later chapter.

UTF 8 Conversion

The UTIL module provides a number of functions to convert between UTF-8 representation, ASCII 7-bit representation and Unicode represenataions. For example, ***StringToCode()*** converts the first UTF-8 character in a string to a Unicode symbol, and ***StringToVector()*** converts a UTF-8 string into a vector of Unicode symbols.

The result of a conversion can be success, out of range, or a two character sequence of the skip character (0xFF) followed by another character.